Collaboration and Competition

You are welcome to use this coding environment to train your agent for the project. Follow the instructions below to get started!

1. Start the Environment

Run the next code cell to install a few packages. This line will take a few minutes to run!

In [3]:

```
!pip -q install ./python
```

The environment is already saved in the Workspace and can be accessed at the file path provided below.

In [4]:

```
"""Required Library Imports"""
from unityagents import UnityEnvironment
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F

import random
import copy
from collections import namedtuple, deque
import matplotlib.pyplot as plt

"""You are welcome to use this coding environment to train your agent for the project.
Follow the instructions below to get started!"""

env = UnityEnvironment(file_name="/data/Tennis_Linux_NoVis/Tennis")
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :

Unity brain name: TennisBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 8
        Number of stacked Vector Observation: 3
        Vector Action space type: continuous
        Vector Action space size (per agent): 2
        Vector Action descriptions: ,
```

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

In [5]:

```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

2. Examine the State and Action Spaces

Run the code cell below to print some information about the environment.

In [6]:

```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)

# examine the state space
states = env_info.vector_observations
state_size = states.shape[1]
print('There are {} agents. Each observes a state with length: {}'.format(states.shape[
0], state_size))
print('The state for the first agent looks like:', states[0])
```

```
Number of agents: 2
Size of each action: 2
There are 2 agents. Each observes a state with length: 24
The state for the first agent looks like: [ 0.          0.          0.
0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
0.
  0.          0.         -6.65278625 -1.5        -0.          0.
  6.83172083  6.         -0.          0.        ]
```

3. Take Random Actions in the Environment

In the next code cell, you will learn how to use the Python API to control the agent and receive feedback from the environment.

Note that **in this coding environment, you will not be able to watch the agents while they are training**, and you should set `train_mode=True` to restart the environment.

In [7]:

```
""" Trial Run """
for i in range(5):                                         # play game for 5 episodes
    env_info = env.reset(train_mode=False)[brain_name]     # reset the environment
    states = env_info.vector_observations                  # get the current state (for
each agent)
    scores = np.zeros(num_agents)                          # initialize the score (for
 each agent)
    while True:
        actions = np.random.randn(num_agents, action_size) # select an action (for each
agent)
        actions = np.clip(actions, -1, 1)                  # all actions between -1 and
1
        env_info = env.step(actions)[brain_name]           # send all actions to tne en
vironment
        next_states = env_info.vector_observations         # get next state (for each a
gent)
        rewards = env_info.rewards                         # get reward (for each agen
t)
        dones = env_info.local_done                        # see if episode finished
        scores += env_info.rewards                         # update the score (for each
agent)
        states = next_states                               # roll over states to next t
ime step
        if np.any(dones):                                  # exit loop if episode finis
hed
            break
    print('Total score (averaged over agents) this episode: {}'.format(np.mean(scores
)))
```

```
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
Total score (averaged over agents) this episode: -0.004999999888241291
```

When finished, you can close the environment.

In [ ]:

```
env.close()
```

4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! A few **important notes**:

- When training the environment, set `train_mode=True`, so that the line for resetting the environment looks like the following:

```
env_info = env.reset(train_mode=True)[brain_name]
```

- To structure your work, you're welcome to work directly in this Jupyter notebook, or you might like to start over with a new file! You can see the list of files in the workspace by clicking on ***Jupyter*** in the top left corner of the notebook.
- In this coding environment, you will not be able to watch the agents while they are training. However, ***after training the agents***, you can download the saved model weights to watch the agents on your own machine!

In [ ]:

```
"""
###Learning Algorithm

To solve this project Multi Agent Deep Deterministic Policy Gradient algorithm was use
d. The details of the algorithm can be found in the paper given by OpenAI: Multi-Agent
 Actor-Critic for Mixed Cooperative-Competitive Environments

The network diagram of MADDPG is as follows:



The network shows two different Actors (Multi-agents) and a single Critic. MADDPG is a
 policy-based method which are well suited for continuous action spaces such as our Ten
nis environment and can learn stochastic policies.

In contrast to DDPG instead of training each agent to learn from its own actions, MADDP
G incorporates actions taken by all the agents. The environment state depends on the ac
tions taken by all agents (collaboration of the tennis players to maximize rewards) so
 if we train an agent using just its own action the policy network does not get enough
 information to come up with a good policy. MADDPG improves upon DDPG by sharing the ac
tions taken by all agents to train each agent.

Actor-Critic Method

Actor-critic methods leverage the strengths of both policy and value based methods.

The Actor uses a policy-based approach and learns how to act by directly estimating the
optimal policy and maximizing reward through gradient ascent. Critic uses a value-based
approach and learns how to estimate the value, the future cumulative reward, of differe
nt state-action pairs. Actor-critic agents are more stable than value-based agents, whi
le requiring fewer training samples than policy-based agents and accelerates the learni
ng process."""
```

In [ ]:

```
"""Model Architecture
# The model for the Actor_Network is as follows:

(fc1) = nn.Linear(48, 256)
(fc2) = nn.Linear(256, 128)
(fc3) = nn.Linear(128, 2)
where (fc1) and (fc2) are followed by ReLU and (fc3) is followed by Tanh activation fun
ctions.

# The model for the Critic_Network is as follows:

(fcs1) = nn.Linear(48, 256)
(fc2) = nn.Linear(256+4, 126)
(fc3) = nn.Linear(126, 1)
where (fcs1) and (fc2) are followed by ReLU activation function

"""
```

In [ ]:

```
"""Hyperparameters

The hyper-parameters used for the Agent model are:


BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 2e-4        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
LEARN_EVERY = 1         # learning timestep interval
LEARN_NUM = 1           # number of learning passes
GAMMA = 0.99            # discount factor
TAU = 7e-2              # for soft update of target parameters
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.12         # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
EPS_START = 5.5                 # initial value for epsilon in noise decay process in A
gent.act()
EPS_EP_END = 250        # episode to end the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay
"""
```

In [ ]:

```
"""PERFORMANCE iMPROVEMENT

# LEARNING RATE WITH 1E-5 OR LESS
# KEEPING LEARING RATE DIFFERENT BETWEEN 2 AGENTS
# INCREASE EPOCHS
# DENSE NETWORK
#Apply following algorithms to compare with MADDPG:PPO,A3C,D4PG

"""
```

In [8]:

```
""" Model Architecture """

def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)

class Actor(nn.Module):
 """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=256, fc2_units=128):

         """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
            Note: Increase Hidden Layers to increase score (Requires Powerfull GPU)
        """

        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size*2, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
 """Build an actor (policy) network that maps states -> actions."""

        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))


class Critic(nn.Module):
     """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=256, fc2_units=128):

    """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """

        super(Critic, self).__init__()
```

```
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size*2, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+(action_size*2), fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
"""Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

In [9]:

```
"""MADDPG Agent"""


"""Hyper Parameters"""

BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 2e-4        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
LEARN_EVERY = 1         # learning timestep interval
LEARN_NUM = 1           # number of learning passes
GAMMA = 0.99            # discount factor
TAU = 7e-2              # for soft update of target parameters
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.12         # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
EPS_START = 5.5                 # initial value for epsilon in noise decay process in A
gent.act()
EPS_EP_END = 250        # episode to end the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


class Agent():
     """Interacts with and learns from the environment"""

    def __init__(self, state_size, action_size, num_agents, random_seed):

         """Initialize an Agent object.
        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            num_agents (int): number of agents
            random_seed (int): random seed
        """

        self.state_size = state_size
        self.action_size = action_size
        self.num_agents = num_agents
        self.seed = random.seed(random_seed)
        self.eps = EPS_START
        self.eps_decay = 1/(EPS_EP_END*LEARN_NUM)  # set decay rate based on epsilon en
d target
        self.timestep = 0

        # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size, random_seed).to(device)
        self.actor_target = Actor(state_size, action_size, random_seed).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

        # Critic Network (w/ Target Network)
        self.critic_local = Critic(state_size, action_size, random_seed).to(device)
        self.critic_target = Critic(state_size, action_size, random_seed).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC
, weight_decay=WEIGHT_DECAY)

        # Noise process
```

```
        self.noise = OUNoise((num_agents, action_size), random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)

    def step(self, state, action, reward, next_state, done, agent_number):
        """Save experience in replay memory, and use random sample from buffer to learn."""

        self.timestep += 1
        # Save experience / reward
        self.memory.add(state, action, reward, next_state, done)
        # Learn, if enough samples are available in memory and at learning interval set
tings
        if len(self.memory) > BATCH_SIZE and self.timestep % LEARN_EVERY == 0:
                for _ in range(LEARN_NUM):
                    experiences = self.memory.sample()
                    self.learn(experiences, GAMMA, agent_number)

    def act(self, states, add_noise):
        """Returns actions for both agents as per current policy, given their respective sta
tes."""

        states = torch.from_numpy(states).float().to(device)
        actions = np.zeros((self.num_agents, self.action_size))
        self.actor_local.eval()
        with torch.no_grad():
            # get action for each agent and concatenate them
            for agent_num, state in enumerate(states):
                action = self.actor_local(state).cpu().data.numpy()
                actions[agent_num, :] = action
        self.actor_local.train()
        # add noise to actions
        if add_noise:
            actions += self.eps * self.noise.sample()
        actions = np.clip(actions, -1, 1)
        return actions

    def reset(self):
        self.noise.reset()

    def learn(self, experiences, gamma, agent_number):
        states, actions, rewards, next_states, dones = experiences

        # ---------------------------- update critic ---------------------------- #
        # Get predicted next-state actions and Q values from target models
        actions_next = self.actor_target(next_states)
        # Construct next actions vector relative to the agent
        if agent_number == 0:
            actions_next = torch.cat((actions_next, actions[:,2:]), dim=1)
        else:
            actions_next = torch.cat((actions[:,:2], actions_next), dim=1)
        # Compute Q targets for current states (y_i)
        Q_targets_next = self.critic_target(next_states, actions_next)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        # Compute critic loss
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
```

```
        self.critic_optimizer.step()

        # ---------------------------- update actor ---------------------------- #
        # Compute actor loss
        actions_pred = self.actor_local(states)
        # Construct action prediction vector relative to each agent
        if agent_number == 0:
            actions_pred = torch.cat((actions_pred, actions[:,2:]), dim=1)
        else:
            actions_pred = torch.cat((actions[:,:2], actions_pred), dim=1)
        # Compute actor loss
        actor_loss = -self.critic_local(states, actions_pred).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

        # update noise decay parameter
        self.eps -= self.eps_decay
        self.eps = max(self.eps, EPS_FINAL)
        self.noise.reset()

    def soft_update(self, local_model, target_model, tau):
        for target_param, local_param in zip(target_model.parameters(), local_model.par
ameters()):
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)

class OUNoise:
 """Ornstein-Uhlenbeck process."""
    def __init__(self, size, seed, mu=0.0, theta=OU_THETA, sigma=OU_SIGMA):
          """Initialize parameters and noise process.
        Params
        ======
            mu (float)     : long-running mean
            theta (float) : speed of mean reversion
            sigma (float) : volatility parameter
        """
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.size = size
        self.reset()

    def reset(self):
     """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.s
ize)
        self.state = x + dx
        return self.state

class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""
```

```
    def __init__(self, action_size, buffer_size, batch_size, seed):
         """Initialize a ReplayBuffer object.
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "rew
ard", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
  """Randomly sample a batch of experiences from memory."""

        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not N
one])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
None])).float().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if
e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not Non
e]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
       """Return the current size of internal memory."""
        return len(self.memory)
```

In [10]:

```
SOLVED_SCORE = 2.7
CONSEC_EPISODES = 100
PRINT_EVERY = 10
ADD_NOISE = True


def maddpg(n_episodes=6000, max_t=1000, train_mode=True):
    """Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

    Params
    ======
        n_episodes (int)      : maximum number of training episodes
        max_t (int)           : maximum number of timesteps per episode
        print_every (int)     : interval to display results
    """

    scores_window = deque(maxlen=CONSEC_EPISODES)
    scores_all = []
    moving_average = []
    best_score = -np.inf
    best_episode = 0
    already_solved = False

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=train_mode)[brain_name]         # reset the env
ironment
        states = np.reshape(env_info.vector_observations, (1,48)) # get states and comb
ine them
        agent_0.reset()
        agent_1.reset()
        scores = np.zeros(num_agents)
        while True:
            actions = get_actions(states, ADD_NOISE)           # choose agent actions a
nd combine them
            env_info = env.step(actions)[brain_name]           # send both agents' acti
ons together to the environment
            next_states = np.reshape(env_info.vector_observations, (1, 48)) # combine t
he agent next states
            rewards = env_info.rewards                         # get reward
            done = env_info.local_done                         # see if episode finishe
d
            agent_0.step(states, actions, rewards[0], next_states, done, 0) # agent 1 l
earns
            agent_1.step(states, actions, rewards[1], next_states, done, 1) # agent 2 l
earns
            scores += np.max(rewards)                          # update the score for e
ach agent
            states = next_states                               # roll over states to ne
xt time step
            if np.any(done):                                   # exit loop if episode f
inished
                break

        ep_best_score = np.max(scores)
        scores_window.append(ep_best_score)
        scores_all.append(ep_best_score)
        moving_average.append(np.mean(scores_window))

        # save best score
```

```
        if ep_best_score > best_score:
            best_score = ep_best_score
            best_episode = i_episode

        # print results
        if i_episode % 10 == 0:
            print('\rEpisodes {:0>4d}-{:0>4d}\tMax Reward: {:.3f}\tMoving Average: {:.3
f}'.format(
                i_episode-PRINT_EVERY, i_episode, np.max(scores_all[-PRINT_EVERY:]), mo
ving_average[-1]))

        if i_episode % 100 == 0:
            print('\r\nEpisodes {:0>4d}-{:0>4d}\tMax Reward: {:.3f}\tMoving Average:
{:.3f} \n Saved!!!\n'.format(
                i_episode-PRINT_EVERY, i_episode, np.max(scores_all[-PRINT_EVERY:]), mo
ving_average[-1]))
            torch.save(agent_0.actor_local.state_dict(), 'checkpoint_actor_0.pth')
            torch.save(agent_0.critic_local.state_dict(), 'checkpoint_critic_0.pth')
            torch.save(agent_1.actor_local.state_dict(), 'checkpoint_actor_1.pth')
            torch.save(agent_1.critic_local.state_dict(), 'checkpoint_critic_1.pth')

        # determine if environment is solved and keep best performing models
        if moving_average[-1] >= 2.5:
            if not already_solved:
                print('\r<-- Environment solved in {:d} episodes! \
                \n<-- Moving Average: {:.3f} over past {:d} episodes'.format(
                    i_episode-CONSEC_EPISODES, moving_average[-1], CONSEC_EPISODES))
                already_solved = True
                # save weights
                torch.save(agent_0.actor_local.state_dict(), 'checkpoint_actor_0.pth')
                torch.save(agent_0.critic_local.state_dict(), 'checkpoint_critic_0.pth'
)
                torch.save(agent_1.actor_local.state_dict(), 'checkpoint_actor_1.pth')
                torch.save(agent_1.critic_local.state_dict(), 'checkpoint_critic_1.pth'
)
            elif ep_best_score >= best_score:
                print('\r<-- Best episode so far!\
                \nEpisode {:0>4d}\tMax Reward: {:.3f}\tMoving Average: {:.3f}'.format(
                i_episode, ep_best_score, moving_average[-1]))
                # save weights
                torch.save(agent_0.actor_local.state_dict(), 'checkpoint_actor_0.pth')
                torch.save(agent_0.critic_local.state_dict(), 'checkpoint_critic_0.pth'
)
                torch.save(agent_1.actor_local.state_dict(), 'checkpoint_actor_1.pth')
                torch.save(agent_1.critic_local.state_dict(), 'checkpoint_critic_1.pth'
)
            elif (moving_average[-1]) >= 2.5:
                # stop training if model stops converging
                break
            else:
                continue

    return scores_all, moving_average

def get_actions(states, add_noise):
    '''gets actions for each agent and then combines them into one array'''
    action_0 = agent_0.act(states, add_noise)    # agent 0 chooses an action
    action_1 = agent_1.act(states, add_noise)    # agent 1 chooses an action
    return np.concatenate((action_0, action_1), axis=0).flatten()
```

In [11]:

```
"""Training  Model """

# initialize agents
agent_0 = Agent(state_size, action_size, num_agents=1, random_seed=0)
agent_1 = Agent(state_size, action_size, num_agents=1, random_seed=0)


# Hyper Parameters loop

BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 2e-4        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
LEARN_EVERY = 1         # learning timestep interval
LEARN_NUM = 1           # number of learning passes
GAMMA = 0.99            # discount factor
TAU = 7e-2              # for soft update of target parameters
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.12         # Ornstein-Uhlenbeck noise parameter, speed of mean reversion
EPS_START = 5.5                 # initial value for epsilon in noise decay process in A
gent.act()
EPS_EP_END = 250        # episode to end the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay

scores, avgs = maddpg()
```

```
Episodes 0000-0010      Max Reward: 0.500       Moving Average: 0.060
Episodes 0010-0020      Max Reward: 0.000       Moving Average: 0.030
Episodes 0020-0030      Max Reward: 0.000       Moving Average: 0.020
Episodes 0030-0040      Max Reward: 0.000       Moving Average: 0.015
Episodes 0040-0050      Max Reward: 0.000       Moving Average: 0.012
Episodes 0050-0060      Max Reward: 0.000       Moving Average: 0.010
Episodes 0060-0070      Max Reward: 0.000       Moving Average: 0.009
Episodes 0070-0080      Max Reward: 0.000       Moving Average: 0.008
Episodes 0080-0090      Max Reward: 0.000       Moving Average: 0.007
Episodes 0090-0100      Max Reward: 0.100       Moving Average: 0.007

Episodes 0090-0100      Max Reward: 0.100       Moving Average: 0.007
 Saved!!!

Episodes 0100-0110      Max Reward: 0.000       Moving Average: 0.001
Episodes 0110-0120      Max Reward: 0.000       Moving Average: 0.001
Episodes 0120-0130      Max Reward: 0.100       Moving Average: 0.002
Episodes 0130-0140      Max Reward: 0.100       Moving Average: 0.003
Episodes 0140-0150      Max Reward: 0.100       Moving Average: 0.006
Episodes 0150-0160      Max Reward: 0.300       Moving Average: 0.012
Episodes 0160-0170      Max Reward: 0.100       Moving Average: 0.014
Episodes 0170-0180      Max Reward: 0.300       Moving Average: 0.017
Episodes 0180-0190      Max Reward: 0.100       Moving Average: 0.019
Episodes 0190-0200      Max Reward: 0.100       Moving Average: 0.020

Episodes 0190-0200      Max Reward: 0.100       Moving Average: 0.020
 Saved!!!

Episodes 0200-0210      Max Reward: 0.300       Moving Average: 0.027
Episodes 0210-0220      Max Reward: 0.400       Moving Average: 0.034
Episodes 0220-0230      Max Reward: 0.100       Moving Average: 0.035
Episodes 0230-0240      Max Reward: 0.100       Moving Average: 0.035
Episodes 0240-0250      Max Reward: 0.100       Moving Average: 0.034
Episodes 0250-0260      Max Reward: 0.100       Moving Average: 0.029
Episodes 0260-0270      Max Reward: 0.400       Moving Average: 0.039
Episodes 0270-0280      Max Reward: 0.200       Moving Average: 0.042
Episodes 0280-0290      Max Reward: 0.300       Moving Average: 0.047
Episodes 0290-0300      Max Reward: 0.300       Moving Average: 0.057

Episodes 0290-0300      Max Reward: 0.300       Moving Average: 0.057
 Saved!!!

Episodes 0300-0310      Max Reward: 0.600       Moving Average: 0.069
Episodes 0310-0320      Max Reward: 0.400       Moving Average: 0.076
Episodes 0320-0330      Max Reward: 0.400       Moving Average: 0.093
Episodes 0330-0340      Max Reward: 0.300       Moving Average: 0.100
Episodes 0340-0350      Max Reward: 0.400       Moving Average: 0.111
Episodes 0350-0360      Max Reward: 0.500       Moving Average: 0.123
Episodes 0360-0370      Max Reward: 0.400       Moving Average: 0.123
Episodes 0370-0380      Max Reward: 0.400       Moving Average: 0.131
Episodes 0380-0390      Max Reward: 0.200       Moving Average: 0.131
Episodes 0390-0400      Max Reward: 0.900       Moving Average: 0.145

Episodes 0390-0400      Max Reward: 0.900       Moving Average: 0.145
 Saved!!!

Episodes 0400-0410      Max Reward: 0.400       Moving Average: 0.142
Episodes 0410-0420      Max Reward: 0.200       Moving Average: 0.137
Episodes 0420-0430      Max Reward: 0.300       Moving Average: 0.130
Episodes 0430-0440      Max Reward: 0.400       Moving Average: 0.138
Episodes 0440-0450      Max Reward: 0.300       Moving Average: 0.140
```

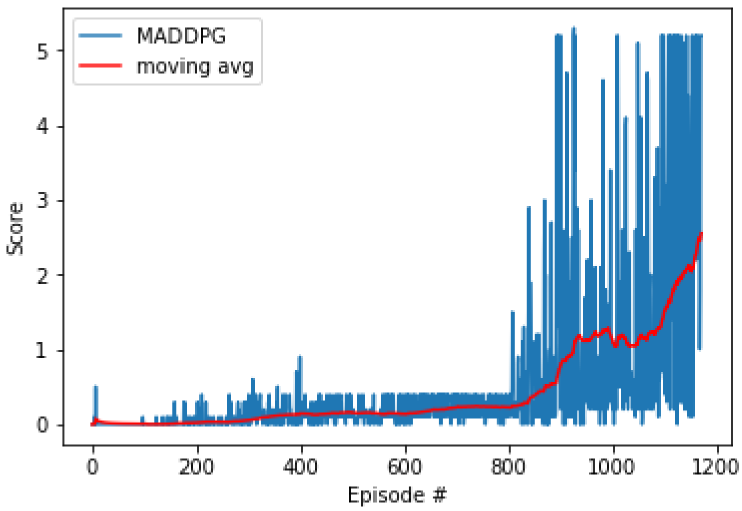
```
Episodes 0450-0460      Max Reward: 0.400       Moving Average: 0.148
Episodes 0460-0470      Max Reward: 0.400       Moving Average: 0.148
Episodes 0470-0480      Max Reward: 0.400       Moving Average: 0.156
Episodes 0480-0490      Max Reward: 0.400       Moving Average: 0.161
Episodes 0490-0500      Max Reward: 0.400       Moving Average: 0.155

Episodes 0490-0500      Max Reward: 0.400       Moving Average: 0.155
 Saved!!!

Episodes 0500-0510      Max Reward: 0.200       Moving Average: 0.149
Episodes 0510-0520      Max Reward: 0.400       Moving Average: 0.155
Episodes 0520-0530      Max Reward: 0.400       Moving Average: 0.155
Episodes 0530-0540      Max Reward: 0.200       Moving Average: 0.150
Episodes 0540-0550      Max Reward: 0.400       Moving Average: 0.146
Episodes 0550-0560      Max Reward: 0.400       Moving Average: 0.137
Episodes 0560-0570      Max Reward: 0.400       Moving Average: 0.149
Episodes 0570-0580      Max Reward: 0.400       Moving Average: 0.150
Episodes 0580-0590      Max Reward: 0.100       Moving Average: 0.142
Episodes 0590-0600      Max Reward: 0.400       Moving Average: 0.140

Episodes 0590-0600      Max Reward: 0.400       Moving Average: 0.140
 Saved!!!

Episodes 0600-0610      Max Reward: 0.400       Moving Average: 0.149
Episodes 0610-0620      Max Reward: 0.400       Moving Average: 0.152
Episodes 0620-0630      Max Reward: 0.400       Moving Average: 0.159
Episodes 0630-0640      Max Reward: 0.400       Moving Average: 0.170
Episodes 0640-0650      Max Reward: 0.400       Moving Average: 0.183
Episodes 0650-0660      Max Reward: 0.400       Moving Average: 0.198
Episodes 0660-0670      Max Reward: 0.400       Moving Average: 0.199
Episodes 0670-0680      Max Reward: 0.400       Moving Average: 0.203
Episodes 0680-0690      Max Reward: 0.400       Moving Average: 0.221
Episodes 0690-0700      Max Reward: 0.400       Moving Average: 0.224

Episodes 0690-0700      Max Reward: 0.400       Moving Average: 0.224
 Saved!!!

Episodes 0700-0710      Max Reward: 0.400       Moving Average: 0.228
Episodes 0710-0720      Max Reward: 0.400       Moving Average: 0.236
Episodes 0720-0730      Max Reward: 0.400       Moving Average: 0.238
Episodes 0730-0740      Max Reward: 0.400       Moving Average: 0.245
Episodes 0740-0750      Max Reward: 0.400       Moving Average: 0.238
Episodes 0750-0760      Max Reward: 0.400       Moving Average: 0.234
Episodes 0760-0770      Max Reward: 0.400       Moving Average: 0.233
Episodes 0770-0780      Max Reward: 0.400       Moving Average: 0.232
Episodes 0780-0790      Max Reward: 0.400       Moving Average: 0.235
Episodes 0790-0800      Max Reward: 0.400       Moving Average: 0.233

Episodes 0790-0800      Max Reward: 0.400       Moving Average: 0.233
 Saved!!!

Episodes 0800-0810      Max Reward: 1.500       Moving Average: 0.249
Episodes 0810-0820      Max Reward: 0.900       Moving Average: 0.254
Episodes 0820-0830      Max Reward: 1.100       Moving Average: 0.273
Episodes 0830-0840      Max Reward: 1.800       Moving Average: 0.304
Episodes 0840-0850      Max Reward: 2.900       Moving Average: 0.380
Episodes 0850-0860      Max Reward: 1.200       Moving Average: 0.437
Episodes 0860-0870      Max Reward: 2.300       Moving Average: 0.461
Episodes 0870-0880      Max Reward: 3.000       Moving Average: 0.510
Episodes 0880-0890      Max Reward: 2.700       Moving Average: 0.542
Episodes 0890-0900      Max Reward: 5.200       Moving Average: 0.746
```

```
Episodes 0890-0900      Max Reward: 5.200       Moving Average: 0.746
 Saved!!!

Episodes 0900-0910      Max Reward: 5.200       Moving Average: 0.846
Episodes 0910-0920      Max Reward: 4.700       Moving Average: 0.905
Episodes 0920-0930      Max Reward: 5.300       Moving Average: 1.115
Episodes 0930-0940      Max Reward: 2.900       Moving Average: 1.162
Episodes 0940-0950      Max Reward: 1.700       Moving Average: 1.122
Episodes 0950-0960      Max Reward: 3.000       Moving Average: 1.147
Episodes 0960-0970      Max Reward: 2.100       Moving Average: 1.218
Episodes 0970-0980      Max Reward: 2.100       Moving Average: 1.219
Episodes 0980-0990      Max Reward: 4.600       Moving Average: 1.257
Episodes 0990-1000      Max Reward: 3.400       Moving Average: 1.144

Episodes 0990-1000      Max Reward: 3.400       Moving Average: 1.144
 Saved!!!

Episodes 1000-1010      Max Reward: 5.200       Moving Average: 1.088
Episodes 1010-1020      Max Reward: 5.200       Moving Average: 1.194
Episodes 1020-1030      Max Reward: 4.100       Moving Average: 1.077
Episodes 1030-1040      Max Reward: 2.300       Moving Average: 1.049
Episodes 1040-1050      Max Reward: 5.100       Moving Average: 1.110
Episodes 1050-1060      Max Reward: 4.100       Moving Average: 1.147
Episodes 1060-1070      Max Reward: 4.700       Moving Average: 1.203
Episodes 1070-1080      Max Reward: 2.000       Moving Average: 1.208
Episodes 1080-1090      Max Reward: 3.700       Moving Average: 1.276
Episodes 1090-1100      Max Reward: 5.200       Moving Average: 1.488

Episodes 1090-1100      Max Reward: 5.200       Moving Average: 1.488
 Saved!!!

Episodes 1100-1110      Max Reward: 5.200       Moving Average: 1.659
Episodes 1110-1120      Max Reward: 5.200       Moving Average: 1.800
Episodes 1120-1130      Max Reward: 5.200       Moving Average: 1.947
Episodes 1130-1140      Max Reward: 5.200       Moving Average: 2.030
Episodes 1140-1150      Max Reward: 5.200       Moving Average: 2.094
Episodes 1150-1160      Max Reward: 5.200       Moving Average: 2.254
Episodes 1160-1170      Max Reward: 5.200       Moving Average: 2.462
<-- Environment solved in 1071 episodes!
<-- Moving Average: 2.505 over past 100 episodes
```

In [13]:

```
""" Agent Training Performance Score """

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='MADDPG')
plt.plot(np.arange(len(scores)), avgs, c='r', label='moving avg')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
fig.show()
fig.savefig("MADDPG_1e04__3000.png")
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/figure.py:418: UserWarni
ng: matplotlib is currently using a non-GUI backend, so cannot show the fi
gure
  "matplotlib is currently using a non-GUI backend, "
```



In [12]:

```
""" Reinitialize the agents (if needed) """
agent_0 = Agent(state_size, action_size, num_agents=1, random_seed=0)
agent_1 = Agent(state_size, action_size, num_agents=1, random_seed=0)

"""  load the weights from file """
agent_0_weights = 'checkpoint_actor_0.pth'
agent_1_weights = 'checkpoint_actor_1.pth'
agent_0.actor_local.load_state_dict(torch.load(agent_0_weights))
agent_1.actor_local.load_state_dict(torch.load(agent_1_weights))
```

In [14]:

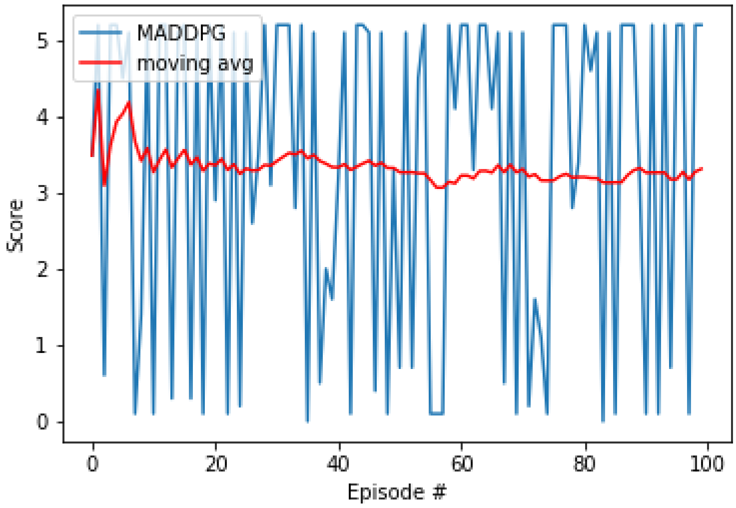
```
"""Testing Model Reesult"""

CONSEC_EPISODES = 50
PRINT_EVERY = 1
ADD_NOISE = False

def test(n_episodes=100, max_t=1000, train_mode=False):

    scores_window = deque(maxlen=CONSEC_EPISODES)
    scores_all = []
    moving_average = []

    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=train_mode)[brain_name]         # reset the environment
        states = np.reshape(env_info.vector_observations, (1,48)) # get states and combine them
        scores = np.zeros(num_agents)
        while True:
            actions = get_actions(states, ADD_NOISE)           # choose agent actions and combine them
            env_info = env.step(actions)[brain_name]           # send both agents' actions together to the environment
            next_states = np.reshape(env_info.vector_observations, (1, 48)) # combine the agent next states
            rewards = env_info.rewards                         # get reward
            done = env_info.local_done                         # see if episode finished
            scores += np.max(rewards)                          # update the score for each agent
            states = next_states                               # roll over states to next time step
            if np.any(done):                                   # exit loop if episode finished
                break

        ep_best_score = np.max(scores)
        scores_window.append(ep_best_score)
        scores_all.append(ep_best_score)
        moving_average.append(np.mean(scores_window))

        # print results
        if i_episode % PRINT_EVERY == 0:
            print('Episodes {:0>4d}-{:0>4d}\tMax Reward: {:.3f}\tMoving Average: {:.3f}'.format(
                i_episode-PRINT_EVERY, i_episode, np.max(scores_all[-PRINT_EVERY:]), moving_average[-1]))

    return scores_all, moving_average
```

In [15]:

```
scores, avgs = test()

# plot the scores
import numpy as np
import random
import time
import torch
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='MADDPG')
plt.plot(np.arange(len(scores)), avgs, c='r', label='moving avg')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
fig.savefig("MADDPG_Test2.png")
fig.show()
```

```
Episodes 0000-0001      Max Reward: 3.490       Moving Average: 3.490
Episodes 0001-0002      Max Reward: 5.200       Moving Average: 4.345
Episodes 0002-0003      Max Reward: 0.600       Moving Average: 3.097
Episodes 0003-0004      Max Reward: 5.200       Moving Average: 3.623
Episodes 0004-0005      Max Reward: 5.200       Moving Average: 3.938
Episodes 0005-0006      Max Reward: 4.500       Moving Average: 4.032
Episodes 0006-0007      Max Reward: 5.100       Moving Average: 4.184
Episodes 0007-0008      Max Reward: 0.100       Moving Average: 3.674
Episodes 0008-0009      Max Reward: 1.400       Moving Average: 3.421
Episodes 0009-0010      Max Reward: 5.100       Moving Average: 3.589
Episodes 0010-0011      Max Reward: 0.100       Moving Average: 3.272
Episodes 0011-0012      Max Reward: 5.200       Moving Average: 3.433
Episodes 0012-0013      Max Reward: 5.200       Moving Average: 3.568
Episodes 0013-0014      Max Reward: 0.300       Moving Average: 3.335
Episodes 0014-0015      Max Reward: 5.100       Moving Average: 3.453
Episodes 0015-0016      Max Reward: 5.200       Moving Average: 3.562
Episodes 0016-0017      Max Reward: 0.300       Moving Average: 3.370
Episodes 0017-0018      Max Reward: 5.100       Moving Average: 3.466
Episodes 0018-0019      Max Reward: 0.100       Moving Average: 3.289
Episodes 0019-0020      Max Reward: 5.200       Moving Average: 3.385
Episodes 0020-0021      Max Reward: 2.900       Moving Average: 3.361
Episodes 0021-0022      Max Reward: 5.200       Moving Average: 3.445
Episodes 0022-0023      Max Reward: 0.100       Moving Average: 3.300
Episodes 0023-0024      Max Reward: 5.100       Moving Average: 3.375
Episodes 0024-0025      Max Reward: 0.200       Moving Average: 3.248
Episodes 0025-0026      Max Reward: 5.100       Moving Average: 3.319
Episodes 0026-0027      Max Reward: 2.600       Moving Average: 3.292
Episodes 0027-0028      Max Reward: 3.500       Moving Average: 3.300
Episodes 0028-0029      Max Reward: 5.200       Moving Average: 3.365
Episodes 0029-0030      Max Reward: 3.100       Moving Average: 3.356
Episodes 0030-0031      Max Reward: 5.200       Moving Average: 3.416
Episodes 0031-0032      Max Reward: 5.200       Moving Average: 3.472
Episodes 0032-0033      Max Reward: 5.200       Moving Average: 3.524
Episodes 0033-0034      Max Reward: 2.800       Moving Average: 3.503
Episodes 0034-0035      Max Reward: 5.200       Moving Average: 3.551
Episodes 0035-0036      Max Reward: 0.000       Moving Average: 3.453
Episodes 0036-0037      Max Reward: 5.100       Moving Average: 3.497
Episodes 0037-0038      Max Reward: 0.500       Moving Average: 3.418
Episodes 0038-0039      Max Reward: 2.000       Moving Average: 3.382
Episodes 0039-0040      Max Reward: 1.600       Moving Average: 3.337
Episodes 0040-0041      Max Reward: 3.200       Moving Average: 3.334
Episodes 0041-0042      Max Reward: 5.100       Moving Average: 3.376
Episodes 0042-0043      Max Reward: 0.100       Moving Average: 3.300
Episodes 0043-0044      Max Reward: 5.200       Moving Average: 3.343
Episodes 0044-0045      Max Reward: 5.200       Moving Average: 3.384
Episodes 0045-0046      Max Reward: 5.100       Moving Average: 3.422
Episodes 0046-0047      Max Reward: 0.400       Moving Average: 3.357
Episodes 0047-0048      Max Reward: 5.100       Moving Average: 3.394
Episodes 0048-0049      Max Reward: 0.100       Moving Average: 3.326
Episodes 0049-0050      Max Reward: 3.300       Moving Average: 3.326
Episodes 0050-0051      Max Reward: 0.700       Moving Average: 3.270
Episodes 0051-0052      Max Reward: 5.100       Moving Average: 3.268
Episodes 0052-0053      Max Reward: 0.700       Moving Average: 3.270
Episodes 0053-0054      Max Reward: 4.500       Moving Average: 3.256
Episodes 0054-0055      Max Reward: 5.200       Moving Average: 3.256
Episodes 0055-0056      Max Reward: 0.100       Moving Average: 3.168
Episodes 0056-0057      Max Reward: 0.100       Moving Average: 3.068
Episodes 0057-0058      Max Reward: 0.100       Moving Average: 3.068
Episodes 0058-0059      Max Reward: 5.200       Moving Average: 3.144
Episodes 0059-0060      Max Reward: 4.100       Moving Average: 3.124
Episodes 0060-0061      Max Reward: 5.200       Moving Average: 3.226
```

```
Episodes 0061-0062      Max Reward: 5.200       Moving Average: 3.226
Episodes 0062-0063      Max Reward: 3.300       Moving Average: 3.188
Episodes 0063-0064      Max Reward: 5.200       Moving Average: 3.286
Episodes 0064-0065      Max Reward: 5.200       Moving Average: 3.288
Episodes 0065-0066      Max Reward: 4.100       Moving Average: 3.266
Episodes 0066-0067      Max Reward: 5.100       Moving Average: 3.362
Episodes 0067-0068      Max Reward: 0.500       Moving Average: 3.270
Episodes 0068-0069      Max Reward: 5.100       Moving Average: 3.370
Episodes 0069-0070      Max Reward: 0.100       Moving Average: 3.268
Episodes 0070-0071      Max Reward: 5.100       Moving Average: 3.312
Episodes 0071-0072      Max Reward: 0.200       Moving Average: 3.212
Episodes 0072-0073      Max Reward: 1.600       Moving Average: 3.242
Episodes 0073-0074      Max Reward: 1.100       Moving Average: 3.162
Episodes 0074-0075      Max Reward: 0.100       Moving Average: 3.160
Episodes 0075-0076      Max Reward: 5.200       Moving Average: 3.162
Episodes 0076-0077      Max Reward: 5.200       Moving Average: 3.214
Episodes 0077-0078      Max Reward: 5.200       Moving Average: 3.248
Episodes 0078-0079      Max Reward: 2.800       Moving Average: 3.200
Episodes 0079-0080      Max Reward: 3.400       Moving Average: 3.206
Episodes 0080-0081      Max Reward: 5.200       Moving Average: 3.206
Episodes 0081-0082      Max Reward: 4.600       Moving Average: 3.194
Episodes 0082-0083      Max Reward: 5.100       Moving Average: 3.192
Episodes 0083-0084      Max Reward: 0.000       Moving Average: 3.136
Episodes 0084-0085      Max Reward: 5.100       Moving Average: 3.134
Episodes 0085-0086      Max Reward: 0.100       Moving Average: 3.136
Episodes 0086-0087      Max Reward: 5.200       Moving Average: 3.138
Episodes 0087-0088      Max Reward: 5.200       Moving Average: 3.232
Episodes 0088-0089      Max Reward: 5.200       Moving Average: 3.296
Episodes 0089-0090      Max Reward: 3.100       Moving Average: 3.326
Episodes 0090-0091      Max Reward: 0.100       Moving Average: 3.264
Episodes 0091-0092      Max Reward: 5.200       Moving Average: 3.266
Episodes 0092-0093      Max Reward: 0.100       Moving Average: 3.266
Episodes 0093-0094      Max Reward: 5.200       Moving Average: 3.266
Episodes 0094-0095      Max Reward: 0.700       Moving Average: 3.176
Episodes 0095-0096      Max Reward: 5.200       Moving Average: 3.178
Episodes 0096-0097      Max Reward: 5.200       Moving Average: 3.274
Episodes 0097-0098      Max Reward: 0.100       Moving Average: 3.174
Episodes 0098-0099      Max Reward: 5.200       Moving Average: 3.276
Episodes 0099-0100      Max Reward: 5.200       Moving Average: 3.314

/opt/conda/lib/python3.6/site-packages/matplotlib/figure.py:418: UserWarni
ng: matplotlib is currently using a non-GUI backend, so cannot show the fi
gure
  "matplotlib is currently using a non-GUI backend, "
```



In [16]:

```
"""Agent Performance Consistency in Testing"""

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='MADDPG')
plt.plot(np.arange(len(scores)), avgs, c='r', label='moving avg')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left')
fig.savefig("MADDPG_Test2.png")
fig.show()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/figure.py:418: UserWarni
ng: matplotlib is currently using a non-GUI backend, so cannot show the fi
gure
  "matplotlib is currently using a non-GUI backend, "
```